SeBS: A Serverless Benchmark Suite for Function-as-a-Service Computing

Marcin Copik
marcin.copik@inf.ethz.ch
ETH Zürich
Switzerland

Grzegorz Kwaśniewski
ETH Zürich
Switzerland

Maciej Besta
ETH Zürich
Switzerland

Michał Podstawski
Future Processing SA
Poland

Torsten Hoefler
ETH Zürich
Switzerland

ABSTRACT

Function-as-a-Service (FaaS) is one of the most promising directions for the future of cloud services, and serverless functions have immediately become a new middleware for building scalable and cost-efficient applications. However, the quickly moving technology hinders reproducibility, and the lack of a standardized benchmarking suite leads to using ad-hoc solutions and microbenchmarks in serverless research, further complicating meta-analysis and comparison of research solutions. To address this challenge, we propose the Serverless Benchmark Suite: the benchmark for FaaS computing that systematically covers a wide spectrum of cloud resources and applications. Our benchmark consists of the specification of representative workloads, the accompanying implementation infrastructure, and the evaluation methodology that facilitates reproducibility and enables interpretability. We demonstrate that the abstract model of a FaaS execution environment ensures the applicability of our benchmark to multiple commercial providers such as AWS, Azure, and Google Cloud. Our work delivers a standardized, reliable, and evolving evaluation methodology of performance, efficiency, scalability, and reliability of middleware FaaS platforms.

CCS CONCEPTS

• **Networks** → **Cloud computing**; • **Computer systems organization** → **Cloud computing**; • **General and reference** → *Performance*; *Metrics*; *Evaluation*; *Measurement*.

KEYWORDS

benchmark, serverless, function-as-a-service, faas

ACM Reference Format:
Marcin Copik, Grzegorz Kwaśniewski, Maciej Besta, Michał Podstawski, and Torsten Hoefler. 2021. SeBS: A Serverless Benchmark Suite for Function-as-a-Service Computing. In *22nd International Middleware Conference (Middleware '21), December 6–10, 2021, Québec city, QC, Canada.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3464298.3476133

Middleware '21, December 6–10, 2021, Québec city, QC, Canada

ACM ISBN 978-1-4503-8534-3/21/12…$15.00
https://doi.org/10.1145/3464298.3476133

User's perspective	
Pay-as-you-go billing	High computing cost
Massive parallelism	Variable performance
Simplified deployment	Vendor lock-in
Architecture agnostic	Black-box platform
Provider's perspective	
Higher machine utilization	Handling heterogeneity
Fine-grained scheduling	Micro-architecture effects

Table 1: Summary of the FaaS model. Quantitative measurements are needed to assess advantages *and* disadvantages.

SeBS implementation: https://github.com/spcl/serverless-benchmarks
SeBS artifact: https://doi.org/10.5281/zenodo.5357597
Extended paper version: https://arxiv.org/abs/2012.14132

1 INTRODUCTION

Clouds changed the computing landscape with the promise of plentiful resources, economy of scale for everyone, and on-demand availability without up-front or long-term commitment. Reported costs are up to 7× lower than that of a traditional in-house server [22]. The deployment of middleware in cloud evolved from a more hardware-oriented *Infrastructure as a Service* (IaaS) to a more software-oriented *Platform as a Service*, where the cloud service provider takes the responsibility of deploying and scaling resources [37]. *Function-as-a-Service* (FaaS) is a recent development towards fine-grained computing and billing, where stateless functions are used to build modular applications without managing infrastructure and incurring costs for unused services.

The flexible FaaS model may be seen as a necessary connection between ever-increasing demands of diverse workloads on the one hand, and huge data centers with specialized hardware on the other. Serverless functions have already become the software glue for building stateful applications [24, 89, 97]. It is already adopted by most major commercial providers, such as AWS Lambda [1], Azure Functions [3], Google Cloud Functions [5], and IBM Cloud Functions [4], marking the future of cloud computing. From a user perspective, it promises more savings and the pay-as-you-go model where only active function invocations are billed, whereas a standard IaaS virtual machine rental incurs costs even when they are idle [94]. From a provider's perspective, the fine-grained execution model enables high machine utilization through efficient scheduling and oversubscription. Table 1 provides an overview of FaaS advantages and issues.

While serverless computing gained significant traction both in industry and academia, many authors raised several important

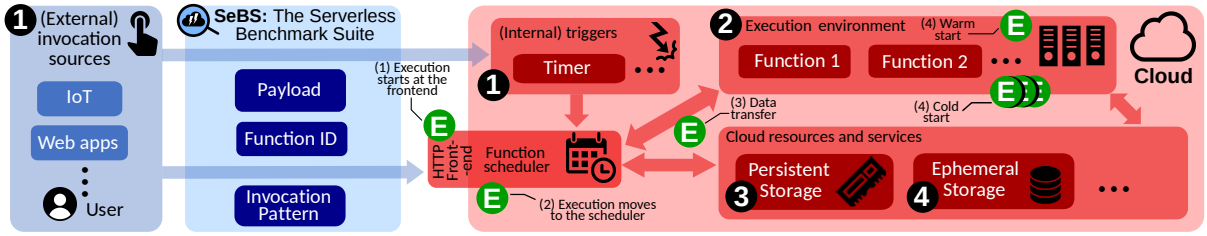


Figure 1: An abstract model of a FaaS platform. Labels: ❶ - triggers, ❷ - execution environment, ❸ - persistent storage, ❹ - ephemeral storage, E - invocation system. Details in Section 2.

issues, for example, vendor lock-in on commercial platforms, lack of standardized tools for development and debugging, unpredictable overheads due to high-latency cold starts [75], and surprisingly high costs of computation-intensive codes [60]. Cost and performance analyses of FaaS applications are further inhibited by the black-box nature of serverless platforms, and existing analyses rarely generalize beyond a given vendor or a tested system. Yet, efficient and scalable software systems cannot be designed without insights into the middleware they're being built upon. Hence, there is an urgent need for a benchmarking design that would (1) specify clear comparison baselines for the evaluation of *many* FaaS workloads on *different* platforms, (2) enable deriving *general* performance, cost, and reliability insights about these evaluations, and (3) facilitate the above with *public and easy to use* implementation.

To address these challenges, we introduce the **Serverless Benchmark Suite (SeBS)**, a FaaS benchmark suite that combines a systematic cover of a wide spectrum of cloud resources with detailed insights into black-box serverless platforms. SeBS comes with a (1) *benchmark specification* based on extensive literature review, (2) a general *FaaS platform model* for wide applicability, (3) a set of *metrics* for effective analysis of cost and performance, (4) *performance models* for generalizing evaluation insights across different cloud infrastructures, and (5) an *implementation* kit that facilitates evaluating existing and future FaaS platforms.

We evaluate SeBS on AWS, Microsoft Azure, and Google Cloud Platform. Overall, with our benchmarks representing a wide variety of real-world workloads, we provide the necessary milestone for serverless functions to become an efficient and reliable software platform for complex and scalable cloud applications.

To summarize, we make the following contributions:

- We propose SeBS, a standardized platform for continuous evaluation, analysis, and comparison of FaaS performance, reliability, and cost-effectiveness.
- We offer novel *metrics* and *experiments* that, among others, enable quantifying the overheads and efficiency of FaaS under various configurations and workloads.
- We provide a full *benchmark implementation* and an open-source *software toolkit* that can automatically build, deploy, and invoke functions on FaaS systems in AWS, Azure, and GCP, three popular cloud providers. The toolkit is modular and can be easily extended to support new FaaS platforms.
- We provide insights into FaaS performance and consistency (Sec. 6, Table 9). We analyze performance and cost overheads of serverless functions, and model cold start patterns and invocation latencies of FaaS platforms.

2 PLATFORM MODEL

We first build a benchmarking model of a FaaS platform that provides an abstraction of key components, see Figure 1. This enables generalizing design details that might vary between providers, or that may simply be unknown due to the black-box nature of a closed-source platform.

❶ **Triggers.** The function lifetime begins with a *trigger*. Cloud providers offer many triggers to express various ways to incorporate functions into a larger application. One example is an HTTP trigger, where every request sent to a function-specific address invokes the function using the passed payload. Such triggers are often used for interactive services or background requests from *Internet-of-Things* (IoT) and edge computing devices. Other triggers are invoked periodically, similarly to cron jobs, or upon events such as a new file upload or a new message in a queue. Finally, functions can be triggered as a part of larger FaaS workflows in dedicated services such as AWS Step and Azure Durable.

❷ **Execution environment.** Functions require a sandbox environment to ensure isolation between tenants. One option is containers, but they may incur overheads of up to 20× over native execution [85]. Another solution is lightweight virtual machines (microVMs). They can provide overheads and bootup times competitive with containers, while improving isolation and security [6, 73].

❸ **Persistent Storage.** The cloud service offers scalable storage and high bandwidth retrieval. Storage offers usually consist of multiple containers known as *buckets* (AWS, Google) and *containers* (Azure). Storage services offer high throughput but also high latency for a low price, with fees in the range of a few cents for 1 GB of data storage, retrieval, or 10,000 write/read operations.

❹ **Ephemeral Storage.** The ephemeral storage service addresses the high latency of persistent storage [66, 67]. Example use cases include storing payload passed between consecutive function invocations [20] and communication in serverless distributed computing [61]. The solutions include scalable, in-memory databases offered by the cloud provider and a custom solution with a VM instance holding an in-memory, key-value storage. While the latter is arguably no longer FaaS and the usage of non-scaling storage services might be considered to be a *serverless anti-pattern* [53], it provides low latency data storage and exchange platform.

E **Invocation System.** The launch process has at least four steps: (a) a cloud endpoint handling the trigger, (b) the FaaS resource manager and scheduler deciding where to place the function instance, (c) communication with the selected cloud server, (d) the server handling invocations with load-balancing and caching. A cold start also adds the execution environment startup latency. These overheads are hidden from the user but understanding them

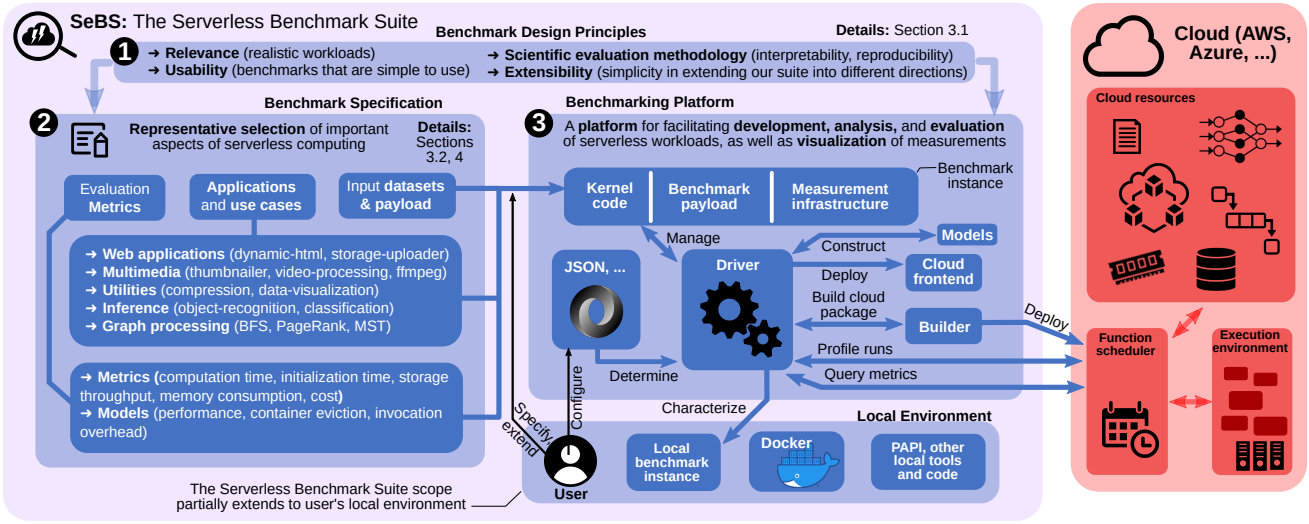


Figure 2: An overview of the offered serverless benchmark suite.

helps to minimize startup latencies. Cloud providers benefit from identifying performance bottlenecks in their systems as well.

3 SERVERLESS MODEL ANALYSIS

To design SeBS, we select candidate FaaS workloads and investigate the fundamental limitations that can throttle the migration of some workloads to the serverless environment.

3.1 Candidate applications

Workloads with a premise for immediate benefits have infrequent invocations, unpredictable and sudden spikes in arriving requests, and fine-grained parallelism. Yet, unprecedented parallelism offered by FaaS is not simple to harness, and many workloads struggle to achieve high performance and suffer from problems such as stragglers [21, 48, 96]. Such FaaS workload classes that may be hard to program for high performance are data analytics [70, 76, 81], distributed compilation [47], video encoding, linear algebra and high-performance computing problems [86], and machine learning training and inference [41, 44, 59].

3.2 FaaS model aspects

Although the adaption of serverless computing is increasing in various domains, the technical peculiarities that made it popular in the first place are now becoming a roadblock for further growth [53, 60]. Both the key advantages and the limitations of FaaS are listed in Table 1. We now describe each aspect to understand the scope of SeBS better.

Computing Cost. FaaS handles infrequent workloads more cost-effectively than persistent VMs. Problems such as machine learning training can be much more expensive than a VM-based solution [48, 60], primarily due to function communication overheads. FaaS burst parallelism outperforms virtual machines in data analytics workloads but inflates costs [76]. Thus, we need to think of computational performance not only in raw FLOP/s but, most importantly, as a FLOP/s per dollar ratio. Here, *SeBS includes cost efficiency as a primary metric to determine the most efficient configuration for a specific workload, analyze the pricing model's flexibility, and compare the costs with IaaS approaches.*

I/O performance. Network I/O affects cold startup latencies, and it is crucial in ephemeral computing as it relies on external storage. As function instances share the bandwidth on a server machine, the co-allocation of functions depending on network bandwidth may degrade performance. Investigation of major cloud providers revealed significant fluctuations of network and I/O performance, with the co-location decreasing throughput up to 20× on AWS [94]. *SeBS includes network and disk performance as a metric to understand I/O requirements of serverless functions better.*

Vendor Lock-In. Lack of standardization in function configuration, deployment, and cloud services complicates development. Each cloud provider requires a customization layer that can be non-trivial. Tackling this, *SeBS provides a transparent library for adapting cloud service interfaces for deployment, invocation, and persistent storage management.*

Heterogeneous Environments. Major FaaS platforms limit user configuration options to the amount of memory allocated and an assigned time to access a virtual CPU. To the best of our knowledge, specialized hardware is only offered by nuclio [17], a data science-oriented and GPU-accelerated FaaS provider. While hardware accelerators are becoming key for scalability [93], serverless functions lack an API to allocate and manage such hardware, similar to solutions in batch systems on HPC clusters [9]. *SeBS includes dedicated tasks that can benefit from specialized hardware.*

Microarchitectural Hardware Effects. The hardware and software stack of server machines is optimized to handle long-running applications, where major performance challenges include high pressure on instruction caches or low counts of instructions per cycle (IPC) [62]. The push to microservices lowers the CPU frontend pressure thanks to a smaller code footprint [50]. Still, they

Policy	AWS	Azure	GCP
Languages (native)	Python, Node.js, C#, Java, C++, and more.	Python, JavaScript, C#, Java etc.	Node.js, Python, Java, Go
Time Limit	15 minutes	10 min / 60 min / Unlimited	9 minutes
Memory Allocation	Static, 128 - 3008 MB	Dynamic, up to 1536 MB	Static, 128, 256, 512, 1024 or 2048 MB
CPU Allocation	Proportional to memory 1 vCPU on 1792 MB	Unknown	Proportional to memory 2.4 GHz CPU at 2048 MB
Billing	Duration and declared memory	Average memory use, duration	Duration, declared CPU and memory
Deployment	zip package up to 250 MB	zip package, Docker image	zip package, up to 100 MB
Concurrency Limit	1000 Functions	200 Function Apss	100 Functions

Table 2: Comparison of major commercial FaaS providers - AWS Lambda [11], Azure Functions [12] and Google Cloud Functions [16]. While commercial services have comparable compute and storage prices, their memory management and billing policies differ fundamentally.

are bounded by single-core performance and frontend inefficiencies due to high instruction cache miss and branch misprediction rate [98]. Serverless functions pose new challenges due to a lack of code and data locality. A microarchitectural analysis of FaaS workloads discovered similar frontend bottlenecks as in microservices: decreased branch predictor performance and increased cache misses due to interfering workloads [85]. *SeBS enables low-level characterization of serverless applications to analyze short-running functions and better understand requirements for an optimal FaaS execution environment.*

3.3 FaaS platforms' limitations

To support the new execution model, cloud providers put restrictions on user's code and resource consumption. Although some of those restrictions can be overcome, developers must design applications with these limitations in mind. Table 2 presents a detailed overview of three commercial Faas platforms: AWS Lambda [1], Azure Functions [3], and Google Cloud Functions [5]. Azure Functions change the semantics with an introduction of *function apps* that consists of multiple functions. The functions are bundled and deployed together, and a single function app instance can use processes and threads to handle multiple function instances from the same app. Thus, they benefit from less frequent cold starts and increased locality while not interfering with isolation and security requirements.

4 BENCHMARK SPECIFICATION

We first discuss principles of reliable benchmarking used in SeBS (Section 4.1). Then, we propose SeBS' specification by analyzing the scope of common FaaS workloads and classifying them into six major categories (Section 4.2).

4.1 Benchmark Design Principles

Designing benchmarks is a difficult "dark art" [45]. For SeBS, we follow well-known guidelines [37, 54, 91].

Relevance. We carefully inspect serverless use cases in the literature to select representative workloads that stress different components of a FaaS platform. We focus on core FaaS components that are widely used on all platforms, and are expected to stay relevant for the foreseeable future.

Usability. Benchmarks that are easy to run benefit from a high degree of self-validation [91]. In addition to a benchmark specification, we provide *a benchmarking platform* and a *reference implementation* to enable automatic deployment and performance evaluation of cloud systems, minimizing the configuration and preparation effort from the user.

Type	Name	Language	Deps
Webapps	dynamic-html	Python	jinja2
		Node.js	mustache
	uploader	Python	-
		Node.js	request
Multimedia	thumbnailer	Python	Pillow
		Node.js	sharp
	video-processing	Python	**ffmpeg**
Utilities	compression	Python	-
	data-vis	Python	squiggle
Inference	image-recognition	Python	pytorch
Scientific	graph-pagerank graph-mst graph-bfs	Python	igraph

Table 3: SeBS applications. One application - *video.processing* - requires a non-pip package: ffmpeg (marked in bold).

Reproducibility & Interpretability. For reproducibility and interpretability of outcomes, we follow established guidelines for scientific benchmarking of parallel codes [55]. We compute the 95% and 99% non-parametric confidence intervals [39, 55] and choose the number of samples such that intervals are within 5% of the median. Still, in multi-tenant systems with shared infrastructure, one cannot exactly reproduce the system state and achieve performance. The FaaS paradigm introduces further challenges with a lack of control on function placement. Thus, in SeBS, we also focus on understanding and minimizing the deviations of measured values. For example, we consider the geolocation of cloud resources and the time of day when running experiments. This enables us to minimize effects such as localized spikes of a cloud activity when many users use it.

Extensibility. While the SeBS implementation uses existing cloud services and relies on interfaces specific to providers, the specification of SeBS depends only on the abstract FaaS model from Section 2. Thus, we do not lock the benchmark in a dependency on a specific commercial system.

4.2 Applications

Our collection of serverless applications is in Table 3. They represent different performance profiles, from simple website backends with minimal CPU overhead to compute-intensive machine learning tasks. To accurately characterize each application's requirements, we conduct a local, non-cloud evaluation of application metrics describing requirements on computing, memory, and external resources (Section 5). The evaluation allows us to classify

applications, verify that our benchmark set is representative, and pick benchmarks according to the required resource consumption.

Web Applications. FaaS platforms allow building simplified static websites where dynamic features can be offloaded to a serverless backend. We include two examples of small but frequently involved functions: *dynamic-html* (dynamic HTML generation from a predefined template) and *storage-uploader* (upload of a file from a given URL to cloud storage). They have low requirements on both CPU and memory.

Multimedia. A common serverless workload is processing multimedia data. Images uploaded require the creation of thumbnails, as we do in our benchmark kernel *thumbnailer*. Videos are usually processed to compress, extract audio, or convert to more suitable formats. We include an application *video-processing* that uses a static build of *ffmpeg* to apply a watermark to a video and convert it to a gif file.

Utilities. Functions are used as backend processing tools for too complex problems for a web server or application frontend. We consider *compression* and *data-vis*. In the former, the function compresses a set of files and returns an archive to the user, as seen in online document office suites and text editors. We use acmart-master template as evaluation input. In the latter, we include the backend of DNAVisualization.org [7, 69], an open-source website providing serverless visualization of DNA sequences, using the squiggle Python library [68]. The website passes DNA data to a function which generates specified visualization and caches results in the storage.

Inference. Serverless functions implement machine learning inference tasks for edge IoT devices and websites to handle scenarios such as image processing with object recognition and classification. We use as an example a standard image recognition with pretrained ResNet-50 model served with the help of pytorch [80] and, for evaluation, images from *fake-resnet* test from MLPerf inference benchmark [82]. Deployment of PyTorch requires additional steps to ensure that the final deployment package meets the limits on the size of the code package. In our case, the most strict requirements are found on AWS Lambda with a limit of 250 megabytes of uncompressed code size. We fix the PyTorch version to 1.0.1 with torchvision in version 0.3. We disable all accelerator support (only CPU), strip shared libraries, and remove tests and binaries from the package. While deep learning frameworks can provide lower inference latency with GPU processing, dedicated accelerators are not currently widely available on FaaS platforms, as discussed in Section 3.2.

Scientific. As an example of scientific workloads, we consider irregular graph computations, a more recent yet established class of workloads [31, 35, 71, 83]. We selected three important problems: Breadth-First Search (BFS) [25, 33], PageRank (PR) [78], and Minimum Spanning Tree (MST). BFS is used in many more complex schemes (e.g., in computing maximum flows [46]), it represents a large family of graph traversal problems [33], and it is a basis of the Graph500 benchmark [77]. PR is a leading scheme for ranking websites and it stands for a class of centrality problems [40, 87]. MST is used in many analytics and engineering problems, and represents graph optimization problems [30, 51, 79]. All three have been extensively research in a past decade [25, 28, 29, 32, 34, 36, 52, 84]. We select the corresponding algorithms such that they are all data-intensive but differ in the details of the workload characteristics (e.g., BFS, unlike PR, may come with severe work imbalance across iterations).

5 BENCHMARK IMPLEMENTATION

We complement the benchmark specification introduced in the previous section with our benchmarking toolkit. We discuss the set of metrics used to characterize application requirements and measure performance overheads (Section 5.1). SeBS enables automatic deployment and invocation of benchmarks, specified in the previous section (Section 5.2). This benchmarking platform is used for parallel experiments that model and analyze the behavior of FaaS systems (Section 6).

5.1 Application Metrics

We now discuss in detail metrics that are measured locally and in the cloud execution.

Local metrics. These metrics provide an accurate profile of application performance and resource usage to the user.

- **Time.** We measure execution time to find which applications require significant computational effort, and we use hardware performance counters to count instructions executed, a metric less likely influenced by system noise.
- **CPU utilization.** We measure the ratio of time spent by the application on the CPU, both in the user and the kernel space, to the wall-clock time. This metric helps to detect applications stalled on external resources.
- **Memory.** Peak memory usage is crucial for determining application configuration and billing. It also enables providers to bound the number of active or suspended containers. Instead of resident set size (RSS) which overapproximates actual memory consumption, we measure the unique set size (USS) and proportional set size (PSS). Thus, we enable an analysis of benefits from page sharing.
- **I/O.** I/O intensive functions may be affected by contention. Average throughput of filesystem I/O and network operations decreases with the number of co-located function invocations that have to share the bandwidth, leading to significant network performance variations [94].
- **Code size.** The size and complexity of dependencies impact the warm *and* cold start latency. Larger code packages increase deployment time from cloud storage and the warm-up time of language runtime.

Cloud metrics. The set of metrics available in the cloud is limited because of the black-box nature of the FaaS system. Still, we can gain additional information through microbenchmarks and modeling experiments (Section 6).

- **Benchmark, Provider and Client Time.** We measure execution time on three levels: directly measure benchmark execution time in cloud, including work performed by function, but not network and system latencies; query cloud provider measurements, adding overheads of language and serverless sandbox; measure end-to-end execution latency on client side, estimating complete overhead with the latency of function scheduling and deployment.

- **Memory.** The actual memory consumption plays a crucial role in determining cost on platforms with dynamic memory allocation. Elsewhere, the peak memory consumption determines the execution settings and billing policies.
- **Cost.** The incurred costs are modeled from billed duration, memory consumption, and a number of requests made to persistent storage. While AWS enables estimating the cost of each function execution, Azure offers a monitoring service with query interval not shorter than one second.

5.2 Implementation

We implement the platform from Figure 2 to fulfill three major requirements: application characterization, deployment to the cloud, and modeling of cloud performance and overheads. We describe SeBS modularity and the support for the inclusion of new benchmarks, metrics, and platforms.

Deployment. SeBS handles all necessary steps of invoking a function in the cloud. We allocate all necessary resources and do not use third-party dependencies, such as the Serverless framework [18], since a flexible and fine-grained control over resources and functions is necessary to efficiently handle large-scale and parallel experiments, e.g., the container eviction model (Sec. 6). For each platform, we implement the simplified interface described below. Furthermore, benchmarks and their dependencies are built within Docker containers resembling function execution workers to ensure binary compatibility with the cloud. Google Cloud Functions use the cloud provider Docker-based build system as required by the provider. *SeBS can be extended with new FaaS platforms by implementing the described interface and specifying Docker builder images.*

```
class FaaS:
 def package_code(directory, language: [Py, JS])
 def create_function(fname, code, lang: [Py, JS]), config)
 def update_function(fname, code, config)
 def create_trigger(fname, type: [SDK, HTTP])
 def query_logs(fname, type: [TIME, MEM, COST])
```

Benchmarks. We use a single benchmark implementation in a high-level language for all cloud providers. Each benchmark includes a Python function to generate inputs for invocations of varying sizes. SeBS implements provider-specific wrappers for entry functions to support different input formats and interfaces. Each benchmark can add custom build actions, including installation of native dependencies and supporting benchmark languages with a custom build process, such as the AWS Lambda C++ Runtime. *New applications integrate easily into SeBS: the user specifies input generation procedure, configures dependencies and optional build actions, and adjusts storage access functionalities.*

```
def function_wrapper(provider_input, provider_env)
 input = json(provider_input)
 start_timer()
 res = function()
 time = end_timer()
 return json(time, statistics(provider_env), res)
```

Storage. We use light-weight wrappers to handle different storage APIs used by cloud providers. Benchmarks use the SeBS abstract storage interface, and we implement one-to-one mappings between our and provider's interface. The overhead is limited to a single redirect of a function call. *New storage solutions require implementing a single interface, and benchmarks will use it automatically.*

Experiments SeBS implements a set of experiments using provided FaaS primitives. Experiments invoke functions through an abstract trigger interface, and we implement cloud SDK and HTTP triggers. The invocation result includes SeBS measurements and an unchanged output of the benchmark application. SeBS metrics are implemented in function wrappers and with the provider log querying facilities. Each experiment includes a postprocessing step that examines execution results and provider logs. *New experiments and triggers are integrated automatically into SeBS through a common interface. SeBS can be extended with new types of metrics by plugging measurement code in SeBS benchmark wrappers, by using the provided log querying facilities, and by returning benchmark-specific measurements directly from the function.*

Technicalities. We use Docker containers with language workers in Python and Node.js in local evaluation; minio [8] implements persistent storage. We use PAPI [90] to gather low-level characteristics (we found the results from Linux *perf* to be unreliable when the application lifetime is short). For cloud metrics, we use provider's API to query execution time, billing, and memory consumption, when available. We use *cURL* to exclude the HTTP connection overheads for client time measurements. We enforce cold starts by updating function configuration on AWS and by publishing a new function version on Azure and GCP.

6 EVALUATION

We show how SeBS provides a consistent and accurate methodology of comparing serverless providers, and assessing the FaaS performance, reliability, and applicability to various classes of workloads. We begin with a local benchmark evaluation to verify that we cover different computing requirements (Section 6.1). Next, we thoroughly evaluate serverless systems' performance-cost tradeoffs (Section 6.2, 6.3). We determine the platforms with the best and most consistent performance and analyze serverless computing's suitability for different types of workloads. Finally, we use SeBS to better understand serverless platforms through performance modeling of invocation latencies (Section 6.4), and container eviction policies (Section 6.5). two major black-box components affecting FaaS applicability as middleware for reliable and scalable applications. We summarize new results and insights provided by SeBS in Table 9.

Configuration We evaluate SeBS on the three most representative FaaS platforms: the default AWS Lambda plan without provisioned concurrency, the standard Linux consumption plan on Azure Functions, and Google Cloud Functions, in regions *us-east-1*, *WestEurope*, and *europe-west1*, respectively. We use S3, Azure Blob Storage, and Google Cloud Storage for persistent storage, HTTP endpoints as function triggers, and deploy Python 3.7 and Node.js 10 benchmarks.

6.1 Benchmark Characteristics

We begin with a local evaluation summarized in Table 4. We selected applications representing different performance profiles, from website backends with minimal CPU overhead and up to compute-intensive machine learning inference. The evaluation allows us to classify applications, verify that our benchmark set is

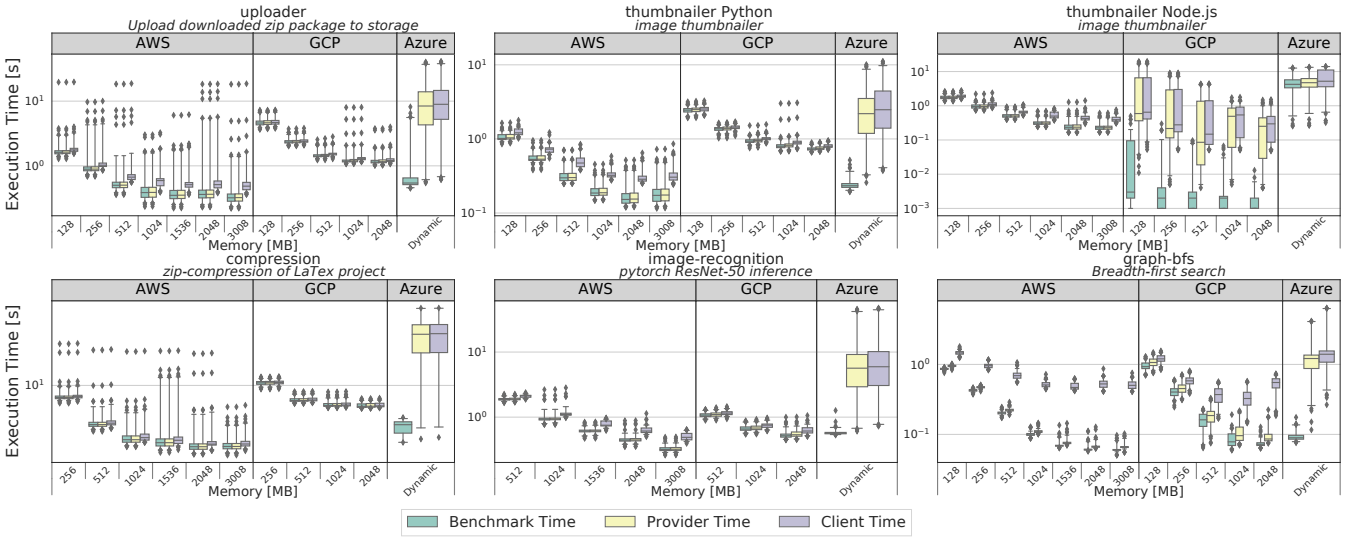


Figure 3: Performance of SeBS applications on AWS Lambda, Azure Functions and Google Cloud Functions. Experiment includes 200 *warm* invocations. Whiskers include data from 2th to 98th percentile.

Name	Lang.	Cold Time [ms]	Warm Time [ms]	Instructions	CPU%
dynamic-html	**P**	130.4 ± 0.7	1.19 ± 0.01	$7.02M \pm 287K$	99.4%
	N	84 ± 2.8	0.28 ± 0.5	-	97.4%
uploader	**P**	236.9 ± 12.7	126.6 ± 8.9	$94.7M \pm 4.45M$	34%
	N	382.8 ± 8.9	135.3 ± 9.6	-	41.7%
thumbnailer	**P**	205 ± 1.4	65 ± 0.8	$404M \pm 293K$	97%
	N	313 ± 4	124.5 ± 4.4	-	98.5%
video-processing	**P**	1596 ± 4.6	1484 ± 5.2	-	-
compression	**P**	607 ± 5.3	470.5 ± 2.8	$1735M \pm 386K$	88.4%
image-recognition	**P**	1268 ± 74	124.8 ± 2.7	$621M \pm 278K$	98.7%
graph-pagerank		194 ± 0.8	106 ± 0.3	$794M \pm 293K$	99%
graph-mst	**P**	125 ± 0.8	38 ± 0.4	$234M \pm 289K$	99%
graph-bfs		123 ± 1.1	36.5 ± 0.5	$222M \pm 300K$	99%

Table 4: Standard characterization of Python and Node.js benchmarks over 50 executions in a local environment on AWS *z1d.metal* machine.

representative and select to experiments benchmarks accordingly to required resource consumption.

6.2 Performance analysis

We design a benchmarking experiment *Perf-Cost* to measure the cost and performance of FaaS executions. We run concurrent function invocations, sampling to obtain N *cold* invocations by enforcing container eviction between each invocations batch. Next, we sample the function executions to obtain N calls to a *warm* container. We measure client, function, and provider time (Section 5.1). We compute non-parametric confidence intervals [56] for client time and select the number of samples $N = 200$ to ensure that intervals are within 5% of the median for AWS while the experiment cost stays negligible. We perform 50 invocations in each batch to include invocations in different sandboxes, and use the same configuration on Azure and GCP for a fair and unbiased comparison of performance and variability. [1]

We benchmark network bandwidth (*uploader*), storage access times and compute performance (*thumbnailer*, *compression*), large cold start deployment and high-memory compute (*image-recognition*), significant output returned (*graph-bfs*), and compare performance across languages (Python and Node.js versions of *thumbnailer*).

Q1 How serverless applications perform on FaaS platforms? Figure 3 presents significant differences in warm invocations between providers, with AWS Lambda providing the best performance on all benchmarks. Each function's execution time decreases until it reaches a plateau associated with sufficient resources to achieve highest observable performance. Only benchmark *graph-bfs* achieves comparable performance on the Google platform, with the largest slowdown observed on benchmarks relying on storage bandwidth (*thumbnailer*, *compression*). On Azure, we note a significant difference between benchmark and provider times on Python benchmarks. To double-check our measurements' correctness and verify if initialization overhead is the source of such discrepancy, we sequentially repeat warm invocations instead of using concurrent benchmark executions. The second batch presents more stable measurements, and we observe performance comparable to AWS on computing benchmarks *image-recognition* and *graph-bfs*.

Our results verify previous findings that CPU and I/O allocation increases with the memory allocation [94]. However, our I/O-bound benchmarks (*uploader*, *compression*) reveal that the distribution of latencies is much wider and includes many outliers, which prevents such functions from achieving consistent and predictable performance.

Conclusions: AWS functions consistently achieve the highest performance. Serverless benefits from larger resource allocation, but selecting the right configuration requires an accurate methodology for measuring short-running functions. I/O-bound workloads are not a great fit for serverless.

Q2 How cold starts affect the performance? We estimate cold startup overheads by considering all N^2 combinations of N cold and N warm measurements. In Figure 4, we summarize the ratios of cold and warm client times of each combination. This

[1] We generate more samples due to unreliable cloud logging services. We always consider first 200 correctly generated samples and don't skip outliers.

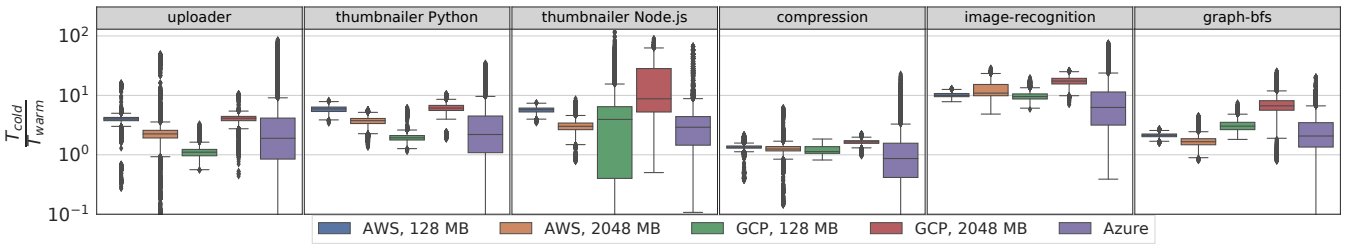


Figure 4: Cold startup overheads of benchmarks on AWS Lambda and Google Cloud Functions, based on *cold* and *warm* executions (Figure 3).

approach doesn't provide a representative usage of Azure Functions, where a single function app instance handles multiple invocations. To estimate real-world cold startups there, instead of *cold* runs, we use concurrent *burst* invocations that include cold and warm executions.

We notice the largest cold startup overheads on benchmark *image-recognition* (large deployment, download model from storage), where a cold execution takes on average up to ten times longer than a warm invocation, which correlates with previous findings (c.f. [74]). Simultaneously, the *compression* benchmark shows that cold start can have a negligible impact for longer running functions (> 10 seconds). Azure provides lower overheads, with the highest gains on benchmarks with a large deployment package and long cold initialization, at the cost of higher variance.

However, we notice an unusual and previously not reported contrast between Amazon and Google platforms: while high memory invocations help to mitigate cold startup overheads on Lambda, providing more CPU allocation for initialization and compilation, they have an adverse effect on Google Functions, except for benchmark *image-recognition* discussed above. A possible explanation of this unexpected result might be a smaller pool of more powerful containers, leading to higher competition between invocations and longer allocation times. *Conclusions: more powerful (and expensive) serverless allocations are not a generic and portable solution to cold startup overheads. Functions with expensive cold initialization benefit from functions apps on Azure.*

Q3 FaaS performance: consistent and portable? Vendor lock-in is a major problem in serverless. We look beyond the usual concern of provider-specific services, and examine changes in function's performance and availability.

Performance deviations In Figure 3, we observe the highest variance in benchmarks relying on I/O bandwidth (*uploader* and *compression*). Compute-intensive applications show consistent execution times (*image-recognition*) while producing a notable number of stragglers on long-running functions (*compression*). Function runtime is not the primary source of variation since we don't observe significant performance differences between Python and Node.js. Google's functions produced fewer outliers on warm invocations. Contrarily, Azure's results present significant performance deviations. Provider and client time measurements were significantly higher and more variant than function time on all benchmarks, except Node.js one, implying that the Python function app generates observed variations. The Node.js benchmark shows a very variable performance, indicating that invocations might be co-located in the same language worker. Finally, we consider the network as a source of variations. The ping latencies to virtual machines allocated in the same resource region as benchmark functions were consistent and equal to 109, 20, and 33 ms on AWS, Azure, and GCP, respectively. Thus, the difference between client and provider times cannot be explained by network latency only.

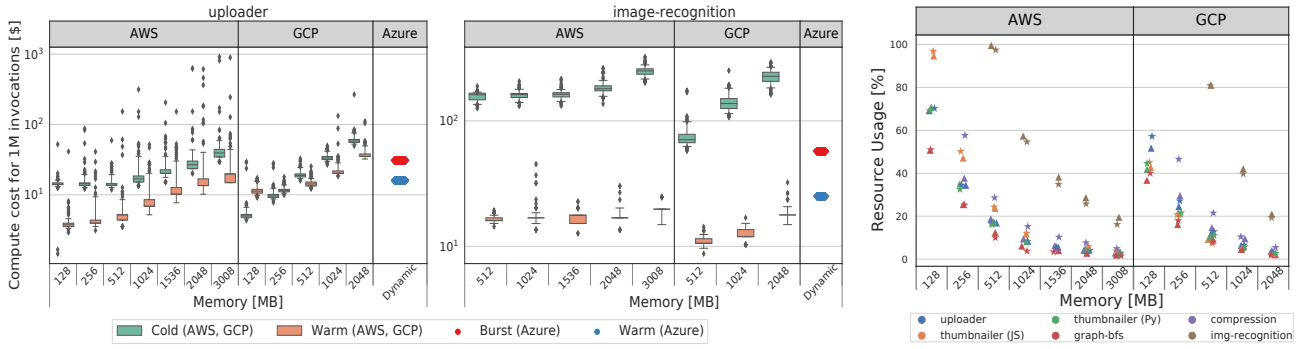
Consistency On AWS, consecutive warm invocations always hit warm containers, even when the number of concurrent calls is large. On the other hand, GCP functions revealed many unexpected cold startups, even if consecutive calls never overlap. The number of active containers can increase up to 100 when processing batches of 50 requests. Possible explanations include slower resources deallocation and a delay in notifying the scheduler about free containers.

Availability Concurrent invocations can fail due to service unavailability, as observed occasionally on Azure and Google Cloud. On the latter, *image-recognition* generated up to 80% error rate on 4096 MB memory when processing 50 invocations, indicating a possible problem with not sufficient cloud resources to process our requests. Similarly, our experiments revealed severe performance degradation on Azure when handling concurrent invocations, as noted in Section 6.2.Q1, with long-running benchmark *compression* being particularly affected. While Azure can deliver an equivalent performance for sequential invocations, it bottlenecks on concurrent invocations of Python functions.

Reliability GCP functions occasionally failed due to exceeding the memory limit, as was the case for benchmarks *image-recognition* and *compression* on 512 MB and 256 MB, respectively. Memory-related failure frequency was 4% and 5.2%, and warm invocations of *compression* had recorded 95th and 99th percentile of memory consumption as 261 and 273 MB, respectively. We didn't observe any issues with the same benchmarks and workload on AWS, where the cloud estimated memory consumption as a maximum of 179 MB and exactly 512 MB, respectively. While the memory allocation techniques could be more lenient on AWS, the GCP function environment might not free resources efficiently.

Conclusions: the performance of serverless functions is not stable, and an identical software configuration does not guarantee portability between FaaS providers. GCP users suffer from much more frequent reliability and availability issues.

Q4 FaaS vs IaaS: is serverless slower? The execution environment of serverless functions brings new sources of overheads [85]. To understand their impact, we compare serverless performance with their natural alternative: virtual machines, where the durable



(a) Compute cost of 1M invocations (USD).

(b) Median ratio of used and billed resources (%).

Figure 5: The cost analysis of performance results from Figure 3: execution cost of 1 million requests (a) and resource usage of cold (▲) and warm (★) executions (b). Azure data is (a) limited to a single average and (b) not available due to limitations of the Azure Monitor systems.

	Upl	Th, Py	Th, JS	Comp	Img-Rec	BFS
IaaS, Local [s]	0.216	0.045	0.166	0.808	0.203	0.03
IaaS, S3 [s]	0.316	0.13	0.191	2.803	0.235	0.03
FaaS [s]	0.389	0.188	0.253	2.949	0.321	0.075
Overhead	1.79x	4.14x	1.43x	3.65x	1.58x	2.49x
Overhead, S3	1.23x	1.43x	1.24x	1.05x	1.37x	2.4x
Mem [MB]	1024	1024	2048	1024	3008	1536

Table 5: Benchmarks performance on AWS Lambda and AWS EC2 t2.micro instance. Median from 200 warm executions.

allocation and higher price provide a more stable environment and data locality. We rent an AWS t2.micro instance with one virtual CPU and 1 GB memory since such instance should have comparable resources with Lambda functions. We deploy SeBS with the local Docker-based execution environment and measure warm execution times of 200 repetitions to estimate latency of constantly warm service. Also, we perform the same experiment with AWS S3 as persistent storage. This provides a more balanced comparison of performance overheads, as cloud provider storage is commonly used instead of a self-deployed storage solution, thanks to its reliability and data durability. We compare the performance against warm provider times (Section 6.2.Q1), selecting configurations where benchmarks obtain high performance and further memory increases don't bring noticeable improvements. We present the summary in Table 5. The overheads of FaaS-ifying the service vary between slightly more than 50% and a slowdown by a factor of four. Equalizing storage access latencies reduces the overheads significantly (Python benchmark *thumbnailer*).

Conclusions: performance overheads of FaaS executions are not uniformly distributed across application classes. The transition from a VM-based deployment to serverless architecture will be accompanied by significant performance losses.

6.3 Cost Analysis

While raw performance may provide valuable insights, the more important question for systems designers is how much does such performance costs. We analyze the cost-effectiveness of results from the *Perf-Cost* experiment described earlier, answering four major research questions.

Q1 How users can optimize the cost of serverless applications? Each provider includes two major fees in the pay-as-you-go billing model: a flat fee for 1 million executions and the cost of consumed compute time and memory, but the implementations are different. AWS charges for reserved memory and computing time rounded up to 100 milliseconds, and GCP has a similar pricing model. On the other hand, Azure allocates memory dynamically and charges for average memory size rounded up to 128 MB.

Since computing and I/O resources are correlated with the amount of requested memory, increasing memory allocation might decrease execution time and a more expensive memory allocation might doesn't necessarily lead to an increase in cost. We study the price of 1 million executions for the I/O-bound *uploader* and compute-bound *image-recognition* benchmarks (Figure 5a). Performance gains are significant for *image-recognition*, where the cost increases negligibly, but the decreased execution time of *compression* does not compensate for growing memory costs. For other benchmarks, we notice a clear cost increase with every expansion of allocated memory. The dynamic allocation on Azure Functions generates higher costs, and they cannot be optimized. *Conclusions: to increase the serverless price-efficiency, the user has to not only characterize the requirements of their application, but the exact performance boundaries - compute, memory, I/O - must be learned as well. Azure Functions generate higher costs because of the dynamic memory allocations.*

Q2 Is the pricing model efficient and fair? FaaS platforms round up execution times and memory consumption, usually to nearest 100 milliseconds and 128 megabytes. Thus, users might be charged for unused duration and resources. With SeBS, we estimate the scale of this problem by comparing actual and billed resource usage. We use the memory consumption of each invocation and the median memory allocation across the experiment on AWS and GCP, respectively. We do not estimate efficiency on Azure because monitor logs contain incorrect information on the memory used[2].

[2]The issues have been reported to Azure team.

			Upl	Th, Py	Th, JS	Comp	Img-Rec	BFS
IaaS	Local	Request/h	16627	79282	21697	4452	17658	119272
	Cloud	Request/h	11371	27503	18819	1284	15312	117153
FaaS		Eco 1M [$]	3.54	2.29	3.75	32.1	15.8	2.08
		Eco B-E	3275	5062	3093	362	733	5568
		Perf 1M [$]	6.67	3.34	10	50	19.58	2.5
		Perf B-E	1740	3480	1160	232	592	4640

Table 6: The break-even point (requests per hour) for the most efficient (Eco) and best performing (Perf) AWS Lambda configuration, compared to IaaS deployment (Table 5). IaaS assumes 100% utilization of the micro.t2 machine costing $0.0116 per hour.

The results in Figure 5b show that the required computing power and I/O bandwidth are not always proportional to memory consumption. Changing the current system would be beneficial to both the user and the provider, who could increase the utilization of servers if declared memory configuration would be closer to actual allocations. Furthermore, rounding up of execution time affects mostly short-running functions, which have gained significant traction as simple processing tools for database and messaging queue events.

Conclusions: memory usage is not necessarily correlated with an allocation of CPU and I/O resources. The current pricing model encourages over-allocation of memory, leading in the end to underutilization of cloud resources.

Q3 FaaS vs IaaS: when is serverless more cost efficient? The most important advantage of serverless functions is the pay-as-you-go model that enables efficient deployment of services handling infrequent workloads. The question arises immediately: how infrequent must be the use of service to achieve lower cost than a dedicated solution with virtual machines? The answer is not immediate since the FaaS environment negatively affects the performance (Section 6.2.Q4). Thus, we attempt the break-even analysis to determine the maximum workload a serverless function can handle in an hour without incurring charges higher than a rental. We summarize in Table 6 the results for the most cost-efficient and the highest performing deployments of our benchmarks on AWS Lambda. While EC2-based solution seems to be a clear cost winner for frequent invocations, its scalability is limited by currently allocated resources. Adding more machines takes time, and multi-core machines introduce additional cost overheads due to underutilization. Serverless functions can scale rapidly and achieve much higher throughput.

Conclusions: the IaaS solution delivers better performance at a lower price, but only if a high utilization is achieved.

Q4 Does the cost differ between providers? Cost estimations of serverless deployments are usually focused on execution time and allocated memory, where fees are quite similar (Section 2, Figure 5a). There are, however, other charges associated with using serverless functions. While storage and logging systems are not strictly required, functions must use the provider's API endpoints to communicate with the outside world. AWS charges a flat fee for an HTTP API but meters each invocation in 512 kB increments [10]. GCP and Azure functions are charged $0.12 and from $0.04 to $0.12, respectively, for each gigabyte of data transferred out [2, 15].

Our benchmark suite includes use cases where sending results directly back to the user is the most efficient way, such as *graph-bfs* returning graph-related data (ca. 78 kB) and *thumbnailer* sending back a processed image (ca. 3 kB). The additional costs for one million invocations can vary from $1 on AWS to almost $9 on Google Cloud and Azure[3].

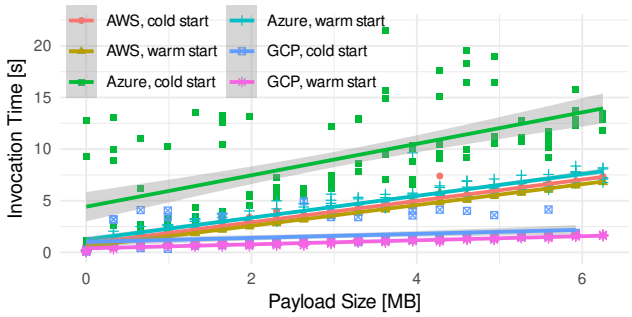


Figure 6: Invocation overhead of functions with varying payload.

Conclusions: billing models of cloud providers include additional charges, and serverless applications communicating a non-negligible amount of data are particularly affected there.

6.4 Invocation overhead analysis

While benchmarking FaaS by saturating bandwidth or compute power tells a part of a story, these services are not designed with such workloads in mind. On the contrary, they are intended to handle large amounts of smaller requests, often arriving in unpredictable behavior, where the latency of starting the system may play a pivotal role. The function invocation latency depends on factors that are hidden from the user (Section 2), and performance results indicated that FaaS systems add non-trivial overheads there (Section 6.2).

As there are no provider-specific APIs to query such metrics, users must estimate these overheads by comparing client-side round-trip latency and function execution time. However, such comparison is meaningful only for symmetric connections. That assumption doesn't hold for serverless: invocation includes the overheads of FaaS controllers, whereas returning results should depend only on network transmission. Instead, we estimate latencies of black-box invocation systems accurately with a different approach in the experiment **Invoc-Overhead**. First, we use timestamps to measure the time that passes between invocation and the execution start, considering all steps, including language worker process overheads. To compare timestamps, we follow an existing clock drift estimation protocol [57]. We measure the invocation latency in regions *us-east-1*, *eastus*, and *us-east1* on AWS, Azure, and GCP, respectively. We analyze round-trip times and discover that they follow an asymmetric distribution, as in [57]. Thus, for clock synchronization, we exchange messages until seeing no lower round-trip time in N consecutive iterations. We pick $N = 10$, since the relative difference between the lowest observable connection time and the minimum time after ten non-decreasing connection times is ca. 5%. Using the benchmarking methodology outlined above, we analyze how the invocation overhead depends on the

[3]HTTP APIs have been available for Lambda since December 2019. REST APIs have higher fees of $3.5 for 1M requests and $0.09 per GB of traffic.

Parameter	Range	Parameter	Range
D_{init}	1-20	ΔT	1-1600 s
Memory	128-1536 MB	Sleep time	1-10 s
Code size	8 kB, 250 MB	Language	Python, Node.js

Table 7: Container eviction experiment parameters.

function input size for 1kB–5.9MB (6MB is the limit for AWS endpoints). The results presented in Figure 6 show the latency cost of cold and warm invocations.

Q1 Is the invocation overhead consistent? We found that invocation latency behavior to be fairly consistent and predictable for cold AWS runs and warm startups on both platforms. At the same time, cold startups on Azure and GCP cannot be easily explained. Similarly to findings in Section 6.2, we observe a cold start behavior that can be caused by unpredictable delays when scheduling functions in the cloud, or the overheads associated with an inefficient implementation of local servers executing the function. *Conclusions: while warm latencies are consistent and predictable, cold startups add unpredictable performance deviations into serverless applications on Azure and GCP.*

Q2 Does the latency change linearly with an increase in payload size? With the exception of Azure's and GCP's cold starts, the latency scales linearly. For warm invocations on AWS, Azure, and GCP, and cold executions on AWS, the linear model fits almost perfectly the measured data, with adjusted R^2 metric 0.99, 0.89, 0.9, and 0.94, respectively. *Conclusions: network transmission times is the only major overhead associated with using large function inputs.*

6.5 Container eviction model.

In the previous section, we observed a significant difference in startup times depending on whether we hit a cold or warm start. Now we analyze how we can increase the chance of hitting warm containers by adjusting the invocation policy. Yet, service providers do not publish their policies. Thus, to guide users, we created the experiment **Eviction-Model** to empirically model function's cold start characteristics.

Q1 Are cold starts deterministic, repeatable, and application agnostic? The container eviction policy can depend on function parameters like number of previous invocations, allocated memory size, execution time, and on global factors: system occupancy or hour. Hence, we use the following benchmarking setup: at a particular time, we submit D_{init} initial invocations, we wait ΔT seconds, and then check how many D_{warm} containers are still active. Next, we test various combinations of D_{init} and ΔT for different function properties (Table 7). Our results reveal that the **AWS** container eviction policy is surprisingly agnostic to many function properties: allocated memory, execution time, language, and code package size. Specifically, after every 380 seconds, half of the existing containers are evicted. The container lifecycles are shown in Figures 7a- 7c.

We also attempted to execute these benchmarks on **Azure** Functions. Yet, massive concurrent invocations led to random and unpredictable failures when invoking functions. *Conclusions: the eviction policy of containers is deterministic and application agnostic, and cold startup frequency can be predicted when scaling serverless applications.*

Reference, Infrastructure	Workloads						Lang.			Platform				Infrastructure			
	Mc	Wb	Ut	Ml	Sc	ML	Py	JS	OT	AW	AZ	GC	OT	Im	At	Dp	Nw
FaaSTest [14]	👍	👎	👎	👎	👎	👍	👍	👍	👍	👍	👍	👍	👎	👍	👍	👎	👎
FaasDom [13, 72]	👍	👎	👎	👎	👎	👎	👍	👍	👍	👍	👍	👍	👎	👍	👍	👎	👍
Somu et al. [88]	👍	👎	👎	👎	👎	👎	👍	👎	👎	👍	👎	👍	👎	👍	👍	👎	👎
EdgeBench [42]	👎	👎	👍	👍	👍	👍	👍	👎	👎	👍	👍	👎	👎	👍	👎	👎	👎
Kim et al. [64, 65]	👍	👍	👍	👍	👍	👍	👍	👎	👎	👍	👍	👍	👎	👍	👎	👎	👎
Serverlessbench [95]	👍	👎	👍	👍	👎	👎	👍	👍	👍	👍	👎	👎	👍	👍	👍	👎	👍
Back et al. [23]	👎	👎	👎	👎	👍	👎	👎	👍	👎	👍	👍	👍	👍	👍	👍	👎	👎
SeBS [This work]	👍	👍	👍	👍	👍	👍	👍	👍	👎	👍	👍	👍	👎	👍	👍	👍	👍

Table 8: Related work analysis: a comparison of existing serverless- and FaaS-related benchmarks, focusing on supported workloads and functionalities (we exclude *proposals not followed by the actual benchmarks* [63, 92]). **Workloads**: **Mc** (microbenchmarks), **Wb** (web applications), **Ut** (utilities), **Ml** (multimedia), **Sc** (scientific), **ML** (Machine learning inference). **Languages**: available implementations, **Py** (Python), **JS** (Node.js), **OT** (Other), **Platform**: **AW** (AWS Lambda), **AZ** (Azure Functions), **GC** (Google Cloud Functions), **OT** (Other commercial/open-source platforms), **Infrastructure**: **Im** (public implementation), **At** (automatic deployment and invocation), **Dp** (building dependencies in compatible environment), **Nw** (delivered new insights or speedups). 👍: Supported. 👍: Partial support. 👎: no support.

Q2 Can we provide simple analytical eviction models? In Equation 1, we provide a simple analytical model of the number of active containers. The model fits the data (Figures 7a-7c) extremely well.

$$D_{warm} = D_{init} \cdot 2^{-p}, \quad p = \lfloor \Delta T / 380s \rfloor \tag{1}$$

We performed a well-established R^2 statistical test to validate its correctness, and the R^2 metric is more than 0.99. The only exception are Python experiments with 10s sleep time, yet even there R^2 is >0.94. Thus, we can use Model 1 to find the time-optimal invocation batch size D_{init}, given that user needs to run n instances of a function with runtime t:

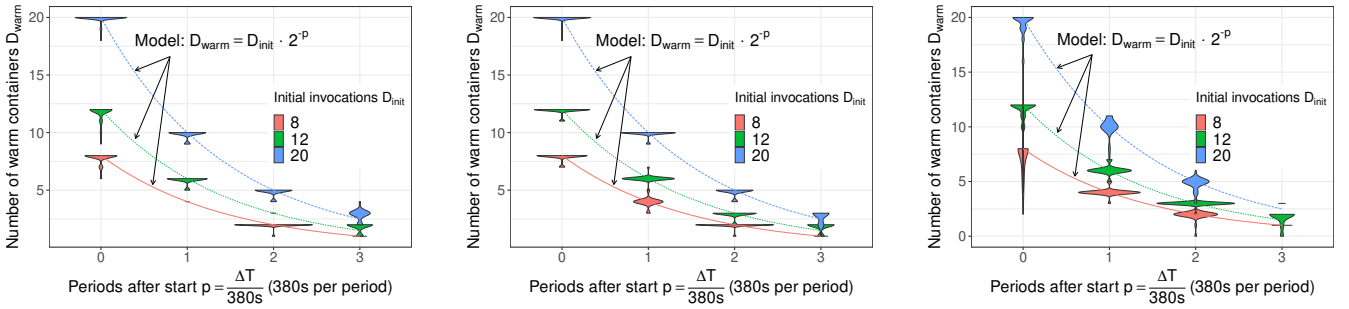
$$D_{init,opt} = n \cdot t / P \tag{2}$$

where $P = 380s$ is the AWS eviction period length.

Conclusions: we derive an analytical model for cold start frequency that can be incorporated into an application to warm up containers and avoid cold starts, without using provisioned concurrency solutions that have a non-serverless billing model.

7 RELATED WORK

We summarize contributions to serverless benchmarking in Table 8. We omit the microservices benchmark suite by Gan et al. [49] as it includes only one FaaS benchmark evaluation. Contrary to many existing cloud benchmarks, SeBS offers a systematic approach where a diverse set of real-world FaaS applications is used instead of microbenchmarks expressing only basic CPU, memory, and I/O requirements. While newer benchmark suites started to adopt automatic deployment and invocation, the installation of dependencies in a compatible way is often omitted when only microbenchmarks are considered. With a vendor-independent benchmark definition and focus on binary compatibility, we enable the straightforward inclusion of new benchmarks representing emerging solutions and usage patterns.



(a) Language: NodeJs, memory allocated: 128 MB, function execution time: 1s.

(b) Language: Python, memory allocated: 1536 MB, function execution time: 1s.

(c) Language: Python, memory allocated: 1536 MB, function execution time: 10s.

Figure 7: Representative scenarios of eviction policies of FaaS containers on AWS.

Results, methods, and insights	Novel insights?
AWS Lambda achieves the best performance on all workloads.	✖[72, 88]
Irregular performance of concurrent Azure Function executions.	✖ [72]
I/O-bound functions experience very high latency variations.	✖ [94]
High-memory allocations increase cold startup overheads on GCP.	👍
GCP functions experience reliability and availability issues.	👍
AWS Lambda performance is not competitive against VMs assuming comparable resources.	👍
High costs of Azure Functions due to unconfigurable deployment.	👍
Resource underutilization due to high granularity of pricing models.	👍
Break-even analysis for IaaS and FaaS deployment.	✖ [76]
The function output size can be a dominating factor in pricing.	👍
Accurate methodology for estimation of invocation latency.	👍
Warm latencies are consistent and depend linearly on payload size.	👍
Highly variable and unpredictable cold latencies on Azure and GCP.	✖ [72]
AWS Lambda container eviction is agnostic to function properties.	✖ [94]
Analytical models of AWS Lambda container eviction policy.	✖ [94]

Table 9: The impact of SeBS: insights and new methods (bolded) provided in our work, with a comparison against similar results in prior work.

SeBS brings new results and insights (Table 9). SeBS goes beyond just an evaluation of the function latency and throughput [23, 64], and we focus on FaaS consistency, efficiency, initialization overheads, and container eviction probabilities and include a set of local metrics that are necessary to characterize resource consumption accurately. Only a single work takes HTTPS overheads into account when measuring invocation latencies [14]. Our work confirms findings that AWS Lambda provides overall the best performance [72, 88], but we obtain this result with a diverse set of practical workloads instead of microbenchmarks. We generalize and extend past results on container eviction modeling [94] and the break-even analysis for a specific application [76]. Maissen et al. [72] report several similar findings to ours. However, their invocation latency estimations depend on round-trip measurements, and they don't take payload size into account. Similarly to us, Yu et al. [95] find other sources of cost inefficiency on AWS. However, their benchmark suite is focused on function composition and white-box, open-source platforms OpenWhisk and Fn. We provide insights into the black-box commercial cloud systems to understand the reliability, and economics of the serverless middleware.

Finally, there are benchmarks in other domains: SPEC [19] (clouds, HPC) LDBC benchmarks (graph analytics) [58], GAPBS [26] and Graph500 [77] (graph related), DaCapo [38] (Java applications), Deep500 [27] and MLPerf [82] (machine learning), and Top500 [43] (dense linear algebra).

8 CONCLUSIONS

We propose SeBS: benchmark suite that facilitates developing, evaluating, and analyzing emerging Function-as-a-Service cloud applications. To design SeBS, we first develop a simple benchmarking model that abstracts away details of cloud components, enabling portability. Then, we deliver a specification and implementation with a wide spectrum of metrics, applications, and performance characteristics.

We evaluated SeBS on the three most popular FaaS providers: AWS, Microsoft Azure, and Google Cloud Platform. Our broad set of results show that (1) AWS is considerably faster in almost all scenarios, (2) Azure suffers from high variance, (3) performance and behavior are not consistent across providers, (4) certain workloads are less suited for serverless deployments and require fine-tuning. We provide new experimental insights, from performance overheads and portability, through cost-efficiency, and developed models for container eviction and invocation overhead. We showed that our open-source benchmark suite gives users an understanding and characterization of the serverless middleware necessary to build and optimize applications using FaaS as its execution backend on the cloud.

ACKNOWLEDGMENTS

This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 programme (grant agreement DAPP, No. 678880, and grant agreement EPIGRAM-HS, No. 801039), and from the Schweizerische Nationalfonds zur Förderung der wissenschaftlichen Forschung (SNF, Swiss National Science Foundation) through Project 170415.

REFERENCES

[1] 2014. AWS Lambda. https://aws.amazon.com/lambda/. Accessed: 2020-01-20.
[2] 2016. Azure: Bandwidth Pricing. https://azure.microsoft.com/en-us/pricing/details/bandwidth/. Accessed: 2020-08-20.
[3] 2016. Azure Functions. https://azure.microsoft.com/en-us/services/functions/. Accessed: 2020-01-20.
[4] 2016. IBM Cloud Functions. https://cloud.ibm.com/functions/. Accessed: 2020-01-20.
[5] 2017. Google Cloud Functions. https://cloud.google.com/functions/. Accessed: 2020-01-20.
[6] 2018. Firecracker. https://github.com/firecracker-microvm/firecracker. Accessed: 2020-01-20.
[7] 2019. DNAvisualization.org. https://github.com/Benjamin-Lee/DNAvisualization.org. Accessed: 2020-01-20.
[8] 2019. MinIO Object Storage. min.io. Accessed: 2020-01-20.
[9] 2019. *SLURM Generic Resource (GRES) Scheduling*. Accessed: 2020-01-20.
[10] 2020. AWS API Pricing. https://aws.amazon.com/api-gateway/pricing/. Accessed: 2020-08-20.
[11] 2020. AWS Lambda Limits. https://docs.aws.amazon.com/lambda/latest/dg/limits.html. Accessed: 2020-01-20.
[12] 2020. Azure Functions scale and hosting. https://docs.microsoft.com/en-us/azure/azure-functions/functions-scale. Accessed: 2020-01-20.
[13] 2020. Faasdom. https://github.com/faas-benchmarking/faasdom. Accessed: 2020-08-01.
[14] 2020. FaaSTest. https://github.com/nuweba/faasbenchmark. Accessed: 2020-08-01.
[15] 2020. Google Cloud Functions Pricing. https://cloud.google.com/functions/pricing. Accessed: 2020-08-20.
[16] 2020. Google Cloud Functions Quotas. https://cloud.google.com/functions/quotas. Accessed: 2020-08-20.
[17] 2020. nuclio. https://nuclio.io/. Accessed: 2020-01-20.
[18] 2020. Serverless Framework. https://github.com/serverless/serverless. Accessed: 2020-08-01.
[19] 2020. Standard Performance Evaluation Corporation (SPEC) Benchmarks. https://www.spec.org/benchmarks.html. Accessed: 2020-08-01.
[20] Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker Hilt. 2018. SAND: Towards High-Performance Serverless Computing. In *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference* (Boston, MA, USA) *(USENIX ATC '18)*. USENIX Association, USA, 923–935.
[21] Lixiang Ao, Liz Izhikevich, Geoffrey M. Voelker, and George Porter. 2018. Sprocket: A Serverless Video Processing Framework. In *Proceedings of the ACM Symposium on Cloud Computing* (Carlsbad, CA, USA) *(SoCC '18)*. Association for Computing Machinery, New York, NY, USA, 263–274. https://doi.org/10.1145/3267809.3267815
[22] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy H. Katz, Andrew Konwinski, Gunho Lee, David A. Patterson, Ariel Rabkin, and Matei Zaharia. 2009. *Above the Clouds: A Berkeley View of Cloud Computing*. Technical Report. EECS Department, University of California, Berkeley.
[23] Timon Back and Vasilios Andrikopoulos. 2018. Using a Microbenchmark to Compare Function as a Service Solutions. In *Service-Oriented and Cloud Computing*. Springer International Publishing, 146–160. https://doi.org/10.1007/978-3-319-99819-0_11
[24] Daniel Barcelona-Pons, Marc Sánchez-Artigas, Gerard París, Pierre Sutra, and Pedro García-López. 2019. On the FaaS Track: Building Stateful Distributed Applications with Serverless Architectures. In *Proceedings of the 20th International Middleware Conference* (Davis, CA, USA) *(Middleware '19)*. Association for Computing Machinery, New York, NY, USA, 41–54. https://doi.org/10.1145/3361525.3361535
[25] Scott Beamer, Krste Asanović, and David Patterson. 2013. Direction-optimizing breadth-first search. *Scientific Programming* 21, 3-4 (2013), 137–148.
[26] Scott Beamer, Krste Asanović, and David Patterson. 2015. The GAP benchmark suite. *arXiv preprint arXiv:1508.03619* (2015).
[27] Tal Ben-Nun, Maciej Besta, Simon Huber, Alexandros Nikolaos Ziogas, Daniel Peter, and Torsten Hoefler. 2019. A modular benchmarking infrastructure for high-performance and reproducible deep learning. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 66–77.
[28] Pavel Berkhin. 2005. A survey on PageRank computing. *Internet mathematics* 2, 1 (2005), 73–120.
[29] Maciej Besta et al. 2019. Slim Graph: Practical Lossy Graph Compression for Approximate Graph Processing, Storage, and Analytics. (2019).
[30] Maciej Besta, Armon Carigiet, Kacper Janda, Zur Vonarburg-Shmaria, Lukas Gianinazzi, and Torsten Hoefler. 2020. High-performance parallel graph coloring with strong guarantees on work, depth, and quality. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–17.
[31] Maciej Besta, Marc Fischer, Tal Ben-Nun, Johannes De Fine Licht, and Torsten Hoefler. 2019. Substream-Centric Maximum Matchings on FPGA. In *ACM/SIGDA FPGA*. 152–161.
[32] Maciej Besta and Torsten Hoefler. 2015. Accelerating irregular computations with hardware transactional memory and active messages. In *ACM HPDC*.
[33] Maciej Besta, Florian Marending, Edgar Solomonik, and Torsten Hoefler. 2017. Slimsell: A vectorizable graph representation for breadth-first search. In *IEEE IPDPS*. 32–41.
[34] Maciej Besta, Emanuel Peter, Robert Gerstenberger, Marc Fischer, Michał Podstawski, Claude Barthels, Gustavo Alonso, and Torsten Hoefler. 2019. Demystifying Graph Databases: Analysis and Taxonomy of Data Organization, System Designs, and Graph Queries. *arXiv preprint arXiv:1910.09017* (2019).
[35] Maciej Besta, Michał Podstawski, Linus Groner, Edgar Solomonik, and Torsten Hoefler. 2017. To push or to pull: On reducing communication and synchronization in graph computations. In *Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing*. 93–104.
[36] Maciej Besta, Dimitri Stanojevic, Tijana Zivic, Jagpreet Singh, Maurice Hoerold, and Torsten Hoefler. 2018. Log (graph): a near-optimal high-performance graph representation.. In *PACT*. 7–1.
[37] Carsten Binnig, Donald Kossmann, Tim Kraska, and Simon Loesing. 2009. How is the Weather Tomorrow?: Towards a Benchmark for the Cloud. In *Proceedings of the Second International Workshop on Testing Database Systems* (Providence, Rhode Island) *(DBTest '09)*. ACM, New York, NY, USA, Article 9, 6 pages. https://doi.org/10.1145/1594156.1594168
[38] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. 2006. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. *SIGPLAN Not.* 41, 10 (Oct. 2006), 169–190. https://doi.org/10.1145/1167515.1167488
[39] Jean-Yves Le Boudec. 2011. *Performance Evaluation of Computer and Communication Systems*. EFPL Press.
[40] Ulrik Brandes and Christian Pich. 2007. Centrality estimation in large networks. *International Journal of Bifurcation and Chaos* 17, 07 (2007), 2303–2318.
[41] Joao Carreira, Pedro Fonseca, Alexey Tumanov, Andrew Zhang, and Randy Katz. 2019. Cirrus: A Serverless Framework for End-to-End ML Workflows. In *Proceedings of the ACM Symposium on Cloud Computing* (Santa Cruz, CA, USA) *(SoCC '19)*. Association for Computing Machinery, New York, NY, USA, 13–24. https://doi.org/10.1145/3357223.3362711
[42] Anirban Das, Stacy Patterson, and Mike Wittie. 2018. EdgeBench: Benchmarking Edge Computing Platforms. In *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)*. IEEE. https://doi.org/10.1109/ucc-companion.2018.00053
[43] Jack J Dongarra, Hans W Meuer, Erich Strohmaier, et al. 1997. TOP500 supercomputer sites. *Supercomputer* 13 (1997), 89–111.
[44] L. Feng, P. Kudva, D. Da Silva, and J. Hu. 2018. Exploring Serverless Computing for Neural Network Training. In *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*. 334–341. https://doi.org/10.1109/CLOUD.2018.00049
[45] Enno Folkerts, Alexander Alexandrov, Kai Sachs, Alexandru Iosup, Volker Markl, and Cafer Tosun. 2013. Benchmarking in the Cloud: What It Should, Can, and Cannot Be. In *Selected Topics in Performance Evaluation and Benchmarking*, Raghunath Nambiar and Meikel Poess (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 173–188.
[46] Lester Randolph Ford and Delbert R Fulkerson. 2009. Maximal flow through a network. In *Classic papers in combinatorics*. Springer, 243–248.
[47] Sadjad Fouladi, Francisco Romero, Dan Iter, Qian Li, Shuvo Chatterjee, Christos Kozyrakis, Matei Zaharia, and Keith Winstein. 2019. From Laptop to Lambda: Outsourcing Everyday Jobs to Thousands of Transient Functional Containers. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, Renton, WA, 475–488. https://www.usenix.org/conference/atc19/presentation/fouladi
[48] Sadjad Fouladi, Riad S. Wahby, Brennan Shacklett, Karthikeyan Vasuki Balasubramaniam, William Zeng, Rahul Bhalerao, Anirudh Sivaraman, George Porter, and Keith Winstein. 2017. Encoding, Fast and Slow: Low-Latency Video Processing Using Thousands of Tiny Threads. In *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation* (Boston, MA, USA) *(NSDI'17)*. USENIX Association, USA, 363–376.
[49] Yu Gan, Brendon Jackson, Kelvin Hu, Meghna Pancholi, Yuan He, Brett Clancy, Chris Colen, Fukang Wen, Catherine Leung, Siyuan Wang, Leon Zaruvinsky, Yanqi Zhang, Mateo Espinosa, Rick Lin, Zhongling Liu, Jake Padilla, Christina Delimitrou, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, and Brian Ritchken. 2019. An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud & Edge Systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems - ASPLOS '19*. ACM Press. https://doi.org/10.1145/3297858.3304013
[50] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, and et al. 2019. An Open-Source Benchmark Suite for Microservices and Their Hardware-Software

Implications for Cloud & Edge Systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (Providence, RI, USA) *(ASPLOS '19)*. Association for Computing Machinery, New York, NY, USA, 3–18. https://doi.org/10.1145/3297858.3304013

[51] Lukas Gianinazzi, Pavel Kalvoda, Alessandro De Palma, Maciej Besta, and Torsten Hoefler. 2018. Communication-avoiding parallel minimum cuts and connected components. In *ACM SIGPLAN Notices*, Vol. 53. ACM, 219–232.

[52] Oleksandr Grygorash, Yan Zhou, and Zach Jorgensen. 2006. Minimum spanning tree based clustering algorithms. In *2006 18th IEEE International Conference on Tools with Artificial Intelligence (ICTAI'06)*. IEEE, 73–81.

[53] Joseph M. Hellerstein, Jose M. Faleiro, Joseph E. Gonzalez, Johann Schleier-Smith, Vikram Sreekanti, Alexey Tumanov, and Chenggang Wu. 2018. Serverless Computing: One Step Forward, Two Steps Back. *CoRR* abs/1812.03651 (2018). arXiv:1812.03651 http://arxiv.org/abs/1812.03651

[54] Torsten Hoefler and Roberto Belli. 2015. Scientific Benchmarking of Parallel Computing Systems. ACM, 73:1–73:12. Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC15).

[55] Torsten Hoefler and Roberto Belli. 2015. Scientific benchmarking of parallel computing systems: twelve ways to tell the masses when reporting performance results. In *Proceedings of the international conference for high performance computing, networking, storage and analysis*. 1–12.

[56] Torsten Hoefler and Roberto Belli. 2015. Scientific Benchmarking of Parallel Computing Systems: Twelve Ways to Tell the Masses When Reporting Performance Results. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Austin, Texas) *(SC '15)*. Association for Computing Machinery, New York, NY, USA, Article 73, 12 pages. https://doi.org/10.1145/2807591.2807644

[57] T. Hoefler, T. Schneider, and A. Lumsdaine. 2008. Accurately measuring collective operations at massive scale. In *2008 IEEE International Symposium on Parallel and Distributed Processing*. 1–8.

[58] Alexandru Iosup, Tim Hegeman, Wing Lung Ngai, Stijn Heldens, Arnau Prat-Pérez, Thomas Manhardto, Hassan Chafio, Mihai Capotă, Narayanan Sundaram, Michael Anderson, et al. 2016. LDBC Graphalytics: A benchmark for large-scale graph analysis on parallel and distributed platforms. *Proceedings of the VLDB Endowment* 9, 13 (2016), 1317–1328.

[59] V. Ishakian, V. Muthusamy, and A. Slominski. 2018. Serving Deep Learning Models in a Serverless Platform. In *2018 IEEE International Conference on Cloud Engineering (IC2E)*. 257–262. https://doi.org/10.1109/IC2E.2018.00052

[60] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Carreira, Karl Krauth, Neeraja Jayant Yadwadkar, Joseph E. Gonzalez, Raluca Ada Popa, Ion Stoica, and David A. Patterson. 2019. Cloud Programming Simplified: A Berkeley View on Serverless Computing. *CoRR* abs/1902.03383 (2019). arXiv:1902.03383 http://arxiv.org/abs/1902.03383

[61] Eric Jonas, Shivaram Venkataraman, Ion Stoica, and Benjamin Recht. 2017. Occupy the Cloud: Distributed Computing for the 99%. *CoRR* abs/1702.04024 (2017). arXiv:1702.04024 http://arxiv.org/abs/1702.04024

[62] Svilen Kanev, Juan Pablo Darago, Kim Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David Brooks. 2015. Profiling a Warehouse-Scale Computer. *SIGARCH Comput. Archit. News* 43, 3S (June 2015), 158–169. https://doi.org/10.1145/2872887.2750392

[63] N. Kaviani and M. Maximilien. 2018. CF Serverless: Attempts at a Benchmark for Serverless Computing. https://docs.google.com/document/d/1e7xTz1P9aPpb0CFZucAAI16Rzef7PWSPLN71pNDa5jg. Accessed: 2020-01-20.

[64] Jeongchul Kim and Kyungyong Lee. 2019. FunctionBench: A Suite of Workloads for Serverless Cloud Function Service. In *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*. IEEE. https://doi.org/10.1109/cloud.2019.00091

[65] Jeongchul Kim and Kyungyong Lee. 2019. Practical Cloud Workloads for Serverless FaaS. In *Proceedings of the ACM Symposium on Cloud Computing - SoCC '19*. ACM Press. https://doi.org/10.1145/3357223.3365439

[66] Ana Klimovic, Yawen Wang, Christos Kozyrakis, Patrick Stuedi, Jonas Pfefferle, and Animesh Trivedi. 2018. Understanding Ephemeral Storage for Serverless Analytics. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, Boston, MA, 789–794. https://www.usenix.org/conference/atc18/presentation/klimovic-serverless

[67] Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis. 2018. Pocket: Elastic Ephemeral Storage for Serverless Analytics. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation* (Carlsbad, CA, USA) *(OSDI'18)*. USENIX Association, USA, 427–444.

[68] Benjamin D Lee. 2018. Squiggle: a user-friendly two-dimensional DNA sequence visualization tool. *Bioinformatics* (sep 2018). https://doi.org/10.1093/bioinformatics/bty807

[69] Benjamin D Lee, Michael A Timony, and Pablo Ruiz. 2019. DNAvisualization.org: a serverless web tool for DNA sequence visualization. *Nucleic Acids Research* 47, W1 (06 2019), W20–W25. https://doi.org/10.1093/nar/gkz404 arXiv:https://academic.oup.com/nar/article-pdf/47/W1/W20/28879727/gkz404.pdf

[70] Pedro García López, Marc Sánchez Artigas, Simon Shillaker, Peter R. Pietzuch, David Breitgand, Gil Vernik, Pierre Sutra, Tristan Tarrant, and Ana Juan Ferrer. 2019. ServerMix: Tradeoffs and Challenges of Serverless Data Analytics. *CoRR* abs/1907.11465 (2019). arXiv:1907.11465 http://arxiv.org/abs/1907.11465

[71] Andrew Lumsdaine, Douglas Gregor, Bruce Hendrickson, and Jonathan Berry. 2007. Challenges in parallel graph processing. *Parallel Processing Letters* 17, 01 (2007), 5–20.

[72] Pascal Maissen, Pascal Felber, Peter Kropf, and Valerio Schiavoni. 2020. FaaSdom. *Proceedings of the 14th ACM International Conference on Distributed and Event-based Systems* (Jul 2020). https://doi.org/10.1145/3401025.3401738

[73] Filipe Manco, Costin Lupu, Florian Schmidt, Jose Mendes, Simon Kuenzer, Sumit Sati, Kenichi Yasukata, Costin Raiciu, and Felipe Huici. 2017. My VM is Lighter (and Safer) than Your Container. In *Proceedings of the 26th Symposium on Operating Systems Principles* (Shanghai, China) *(SOSP '17)*. Association for Computing Machinery, New York, NY, USA, 218–233. https://doi.org/10.1145/3132747.3132763

[74] Johannes Manner, Martin EndreB, Tobias Heckel, and Guido Wirtz. 2018. Cold Start Influencing Factors in Function as a Service. *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)* (2018), 181–188.

[75] Johannes Manner, Martin Endreß, Tobias Heckel, and Guido Wirtz. 2018. Cold start influencing factors in function as a service. In *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)*. IEEE, 181–188.

[76] Ingo Müller, Renato Marroquin, and Gustavo Alonso. 2019. Lambada: Interactive Data Analytics on Cold Data using Serverless Cloud Infrastructure. *ArXiv* abs/1912.00937 (2019).

[77] Richard C Murphy, Kyle B Wheeler, Brian W Barrett, and James A Ang. 2010. Introducing the graph 500. *Cray Users Group (CUG)* 19 (2010), 45–74.

[78] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. 1999. *The pagerank citation ranking: Bringing order to the web.* Technical Report. Stanford InfoLab.

[79] Christos H Papadimitriou and Kenneth Steiglitz. 1998. *Combinatorial optimization: algorithms and complexity*. Courier Corporation.

[80] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett (Eds.). Curran Associates, Inc., 8024–8035. http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf

[81] Qifan Pu, Shivaram Venkataraman, and Ion Stoica. 2019. Shuffling, Fast and Slow: Scalable Analytics on Serverless Infrastructure. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. USENIX Association, Boston, MA, 193–206. https://www.usenix.org/conference/nsdi19/presentation/pu

[82] Vijay Janapa Reddi, Christine Cheng, David Kanter, Peter Mattson, Guenther Schmuelling, Carole-Jean Wu, Brian Anderson, Maximilien Breughe, Mark Charlebois, William Chou, Ramesh Chukka, Cody Coleman, Sam Davis, Pan Deng, Greg Diamos, Jared Duke, Dave Fick, J. Scott Gardner, Itay Hubara, Sachin Idgunji, Thomas B. Jablin, Jeff Jiao, Tom St. John, Pankaj Kanwar, David Lee, Jeffery Liao, Anton Lokhmotov, Francisco Massa, Peng Meng, Paulius Micikevicius, Colin Osborne, Gennady Pekhimenko, Arun Tejusve Raghunath Rajan, Dilip Sequeira, Ashish Sirasao, Fei Sun, Hanlin Tang, Michael Thomson, Frank Wei, Ephrem Wu, Lingjie Xu, Koichi Yamada, Bing Yu, George Yuan, Aaron Zhong, Peizhao Zhang, and Yuchen Zhou. 2019. MLPerf Inference Benchmark. arXiv:1911.02549 [cs.LG]

[83] Sherif Sakr, Angela Bonifati, Hannes Voigt, Alexandru Iosup, Khaled Ammar, Renzo Angles, Walid Aref, Marcelo Arenas, Maciej Besta, Peter A Boncz, et al. 2020. The Future is Big Graphs! A Community View on Graph Processing Systems. *arXiv preprint arXiv:2012.06171* (2020).

[84] Hermann Schweizer, Maciej Besta, and Torsten Hoefler. 2015. Evaluating the cost of atomic operations on modern architectures. In *IEEE PACT*. 445–456.

[85] Mohammad Shahrad, Jonathan Balkind, and David Wentzlaff. 2019. Architectural Implications of Function-as-a-Service Computing. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture* (Columbus, OH, USA) *(MICRO '52)*. Association for Computing Machinery, New York, NY, USA, 1063–1075. https://doi.org/10.1145/3352460.3358296

[86] Vaishaal Shankar, Karl Krauth, Qifan Pu, Eric Jonas, Shivaram Venkataraman, Ion Stoica, Benjamin Recht, and Jonathan Ragan-Kelley. 2018. numpywren: serverless linear algebra. *CoRR* abs/1810.09679 (2018). arXiv:1810.09679 http://arxiv.org/abs/1810.09679

[87] Edgar Solomonik, Maciej Besta, Flavio Vella, and Torsten Hoefler. 2017. Scaling betweenness centrality using communication-efficient sparse matrix multiplication. In *ACM/IEEE Supercomputing*. 47.

[88] N. Somu, N. Daw, U. Bellur, and P. Kulkarni. 2020. PanOpticon: A Comprehensive Benchmarking Tool for Serverless Applications. In *2020 International Conference on COMmunication Systems NETworkS (COMSNETS)*. 144–151.

[89] Vikram Sreekanti, Chenggang Wu, Xiayue Charles Lin, Johann Schleier-Smith, Joseph E. Gonzalez, Joseph M. Hellerstein, and Alexey Tumanov. 2020. Cloudburst: Stateful Functions-as-a-Service. *Proc. VLDB Endow.* 13, 12 (July 2020), 2438–2452. https://doi.org/10.14778/3407790.3407836

[90] Dan Terpstra, Heike Jagode, Haihang You, and Jack Dongarra. 2010. Collecting Performance Data with PAPI-C. In *Tools for High Performance Computing 2009*, Matthias S. Müller, Michael M. Resch, Alexander Schulz, and Wolfgang E. Nagel (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 157–173.

[91] Jóakim v. Kistowski, Jeremy A. Arnold, Karl Huppler, Klaus-Dieter Lange, John L. Henning, and Paul Cao. 2015. How to Build a Benchmark. In *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering* (Austin, Texas, USA) *(ICPE '15)*. ACM, New York, NY, USA, 333–336. https://doi.org/10.1145/2668930.2688819

[92] Erwin van Eyk, Alexandru Iosup, Cristina L. Abad, Johannes Grohmann, and Simon Eismann. 2018. A SPEC RG Cloud Group's Vision on the Performance Challenges of FaaS Cloud Architectures. In *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering - ICPE '18*. ACM Press. https://doi.org/10.1145/3185768.3186308

[93] Jeffrey S Vetter, Ron Brightwell, Maya Gokhale, Pat McCormick, Rob Ross, John Shalf, Katie Antypas, David Donofrio, Travis Humble, Catherine Schuman, et al. 2019. *Extreme Heterogeneity 2018-Productive Computational Science in the Era of Extreme Heterogeneity: Report for DOE ASCR Workshop on Extreme Heterogeneity.* Technical Report. Lawrence Berkeley National Lab.(LBNL), Berkeley, CA (United States).

[94] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. 2018. Peeking behind the Curtains of Serverless Platforms. In *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference* (Boston, MA, USA) *(USENIX ATC '18)*. USENIX Association, USA, 133–145.

[95] Tianyi Yu, Qingyuan Liu, Dong Du, Yubin Xia, Binyu Zang, Ziqian Lu, Pingchao Yang, Chenggang Qin, and Haibo Chen. 2020. Characterizing Serverless Platforms with Serverlessbench. In *Proceedings of the 11th ACM Symposium on Cloud Computing* (Virtual Event, USA) *(SoCC '20)*. Association for Computing Machinery, New York, NY, USA, 30–44. https://doi.org/10.1145/3419111.3421280

[96] Miao Zhang, Yifei Zhu, Cong Zhang, and Jiangchuan Liu. 2019. Video Processing with Serverless Computing: A Measurement Study. In *Proceedings of the 29th ACM Workshop on Network and Operating Systems Support for Digital Audio and Video* (Amherst, Massachusetts) *(NOSSDAV '19)*. Association for Computing Machinery, New York, NY, USA, 61–66. https://doi.org/10.1145/3304112.3325608

[97] Wen Zhang, Vivian Fang, Aurojit Panda, and Scott Shenker. 2020. Kappa: A Programming Framework for Serverless Computing. In *Proceedings of the 11th ACM Symposium on Cloud Computing* (Virtual Event, USA) *(SoCC '20)*. Association for Computing Machinery, New York, NY, USA, 328–343. https://doi.org/10.1145/3419111.3421277

[98] Y. Zhu, D. Richins, M. Halpern, and V. J. Reddi. 2015. Microarchitectural implications of event-driven server-side web applications. In *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 762–774. https://doi.org/10.1145/2830772.2830792